Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution

K. Vikram
Cornell University
Ithaca, NY
kvikram@cs.cornell.edu

Abhishek Prateek
IIT Delhi
New Delhi, India
abhishekprateek@gmail.com

Benjamin Livshits
Microsoft Research
Redmond, WA
livshits@microsoft.com

ABSTRACT

Rich Internet applications are becoming increasingly distributed, as demonstrated by the popularity of AJAX or Web 2.0 applications such as Facebook, Google Maps, Hotmail and many others. A typical multi-tier AJAX application consists, at the least, of a server-side component implemented in Java J2EE, PHP or ASP.NET and a client-side component running JavaScript. The resulting application is more responsive because computation has moved closer to the client, avoiding unnecessary network round trips for frequent user actions.

However, once a portion of the code has moved to the client, a malicious user can subvert the client side of the computation, jeopardizing the integrity of the server-side state. In this paper we propose RIPLEY, a system that uses replicated execution to *automatically* preserve the integrity of a distributed computation. RIPLEY replicates a copy of the client-side computation on the trusted server tier. Every client-side event is transferred to the replica of the client for execution. RIPLEY observes results of the computation, both as computed on the client-side and on the server side using the replica of the client-side code. Any discrepancy is flagged as a potential violation of computational integrity.

We built RIPLEY on top of Volta, a *distributing compiler* that translates .NET applications into JavaScript, effectively providing a measure of security by construction for Volta applications. We have evaluated the RIPLEY approach on five representative AJAX applications built in Volta and also on Hotmail, a large widely-used AJAX application. Our results so far suggest that RIPLEY provides a promising strategy for building secure distributed web applications, which places minimal burden on the application developer at the cost of a low performance overhead.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*invasive software*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*;
D.1.2 [**Programming Techniques**]: Automatic Programming—*program transformation, program modification*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'09, November 9–13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-352-5/09/11 ...$10.00.

General Terms

Security, Languages, Design

Keywords

Replication, Tier-splitting, Web applications

1. INTRODUCTION

Web applications are becoming increasingly distributed, marked by the emergence of popular AJAX (asynchronous JavaScript and XML) applications such as Facebook, Google Maps, Hotmail, and many others. A typical multi-tier AJAX application consists of a server tier implemented in Java J2EE or Microsoft .NET and a client tier executing in JavaScript in the browser. The resulting application is more *responsive*, because computation is moved closer to the client, thus avoiding unnecessary network round trips. Network bandwidth is typically also saved for long-running applications because the client can make fine-grained data requests instead of downloading entire HTML files [3]. Unlike a computation performed entirely on the server, however, once a portion of the code is moved to the client tier, the overall computation can no longer be trusted. Indeed, a malicious client can easily manipulate both the data that resides on *and* the code that runs within the browser using one of many readily available proxy- or browser-based data tampering or debugging tools [27, 29].

The application developer is typically responsible for manually splitting the application in a way that places all security-sensitive operations on the server. This is not an easy task because it requires reasoning about the information flow throughout the application, an inherently global property that is hard to establish, especially by manual inspection. While some language-based approaches have recently been proposed to help reason about security [7, 8], these techniques still require a great deal of developer involvement because code needs to be heavily annotated, making these techniques difficult to use for large-scale code bases. For instance, one such approach requires about 20–30% of program lines to be annotated [8]. Moreover, these techniques are challenging to retrofit into existing un-annotated code.

At the same time, manually validating the results computed on the untrusted client tier, while maintaining responsiveness, is a very tough approach to get right. Ensuring that these validation checks are consistently performed on the client (to provide quick feedback to the user) and on the server (to ensure integrity of validation) has been challenging, as evidenced by the frequency of cross-site scripting attacks that are the result of failing to properly sanitize or filter all user input consistently [23, 26, 31, 34, 52].

The problem is made much more challenging if the client and server are expected to exchange arbitrary data structures, whose in-

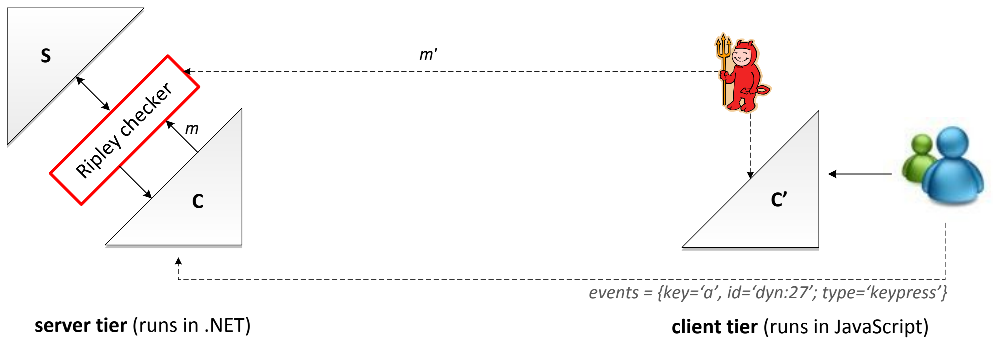
S
Ripley checker
m
C
m'
C'
events = {key='a', id='dyn:27'; type='keypress'}
server tier (runs in .NET)
client tier (runs in JavaScript)

Figure 1: Architecture of RIPLEY: user events are delivered to both the JavaScript client-side component C' and its server-side replica C. RPCs m and m' arriving at the server component S are compared by the RIPLEY checker.

tegrity needs to be checked by the server code after each exchange. The integrity of computation performed by the client code also needs to be validated. Imagine a client-server email application such as Outlook or Hotmail. Suppose the user clicks an image next to an email message to mark it as spam. As a result, the email message is moved to the spam folder and the underlying data structures such as folder sizes, are updated. Note that a *single event* leads to multiple complex operations on the underlying server-side email database: the email has to be removed to one folder, added to another, the counters need to be updated, rules run, etc., the integrity checks for which would be scattered across the server code.

While a robust network protocol can be manually designed, in general, checking that all these actions have been performed properly and the data structures are updated consistently is not a simple task. Moreover, if the distributed application is produced using a *distributing compiler* such as Links [11] or Volta [36], these integrity assertions are difficult to synthesize as the partitioning of the program may often change as a result of application profiling or deployment environment (e.g. desktop vs. mobile devices), thus invalidating old integrity checks. A distributing compiler automatically partitions the application to be run on multiple machines, often in different runtime environments, such as JavaScript and .NET.

Similarly, if an online maze game maintains partial game state and performs logical checks on the client, such as, can a user move to the right, given the current state of the maze and the user's location within it, such checks can be circumvented by a malicious user. Re-executing these checks on the server offers a reliable and conceptually straightforward model for ensuring integrity of server-side state, no matter how the application is split across tiers.

If it is our ultimate (data integrity) goal to maintain the server-side email data store in a consistent state, a reliable way to validate the integrity of this computation is to repeat it on the server side by replaying the mouse click that initiates the action. RIPLEY effectively *restores* the level of security that has been lost by moving a portion of an application to the client. Note that RIPLEY does not try to enhance the security beyond that: a SQL injection or a cross-site scripting vulnerability in the original application will persist in the distributed version; reliance on RIPLEY does not negate the need for input sanitization.

Architecture of RIPLEY**.** In this paper we propose RIPLEY, a system that uses replicated execution to automatically preserve the *integrity* of a distributed computation, such as a typical AJAX application. The architecture of RIPLEY is shown in Figure 1. RIPLEY replicates the client-side computation on the trusted server tier. Every user-initiated event is transferred to the replica of the client for execution. RIPLEY compares results of the computation, both as computed on the client-side and on the server side using the replica of the client-side code. Any discrepancy is flagged as a potential violation of computational integrity.

Our secondary goal is to relieve the application developer of the burden of ensuring distributed application integrity. RIPLEY automatically provides the developer-intended protection for the application without requiring the developer to reason about code placement and trust implications. In line with our security by construction vision, RIPLEY can be fully integrated with the server cloud so that replicated deployment is done fully automatically, as a matter of deployment policy[1]. RIPLEY performs the following key steps:

1. **Capture user events**: RIPLEY augments the client to capture user events within the browser.
2. **Transmit events to the server for replay:** The client runtime is modified to transmit user events to the client's replica C for replay.
3. **Compare server and client results:** The server component S is augmented with a RIPLEY *checker* that compares arriving RPCs m' and m received from the client C' and server-based client replica C, respectively, looking for discrepancies.

These steps are described in detail in Section 3. In summary, RIPLEY relies on re-execution[2] to produce the correct result within replica C based on user events that it receives, effectively ignoring malicious data and code changes that occur on the client. RIPLEY does not rely on the integrity of the client computation. Just as with Web 1.0 applications, the basic assumption throughout this paper is that anything executing on the server tier is believed to be un-compromised and trusted, whereas the client tier, including the browser itself, may be compromised. If the malicious changes result in different RPCs issued to the server, RIPLEY will flag a potential exploit and terminate that client's connection.

As shown in Figure 2, with RIPLEY, a distributed web application can combine the best of both worlds: the application is still responsive because of client-side execution, but the results of this

[1]Note that RIPLEY is primarily designed to protect the integrity of distributed applications. RIPLEY does not remove the need for input validation nor does it eliminate confidentiality concerns. Confidentiality and input validation are important orthogonal issues addressed by prior work [17, 26, 31, 40, 41, 52]. Moreover, while this is not a panacea, for many applications that expect users to authenticate, confidentiality is often addressed through the use of roles, which essentially propagates login credentials to the database tier, in many settings limiting the potential for information leaks.

[2]Re-execution or *replay*, hence the name RIPLEY.

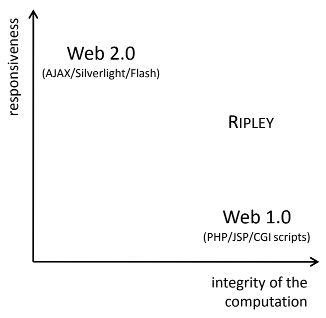

Figure 2: An informal illustration of the responsiveness vs. integrity trade-off for the two dominant web application execution models. RIPLEY **aims to combine the best of both.**

execution do not have to be trusted because they are replayed on the server. In other words, the integrity assurance offered by the application is the same as if it had been run entirely on the server.

Most existing approaches to ensuring integrity of client computation involve the client sending a proof of certain properties that its execution state holds. The server efficiently validates these proofs convincing itself of the integrity of the client execution. For instance, the client could periodically send over its stack traces to the server, and the server could check the traces for any properties it desires. Or the server could build a model of proper client behavior and flag behavioral violations of this model [16]. While they are valuable, these techniques only provide a partial enforcement of integrity.

Feasibility of our approach. In general, replicating the client-side component running in JavaScript on the server is far from easy. We discuss the issues involved in creating a deterministic replay system [6, 39] in detail in Section 5. While the ideas proposed in this paper apply in the context of regular AJAX applications, to simplify the process of replication, RIPLEY capitalizes on a recent trend towards distributing compilers such as Links [11], Hilda [54], Hop [46], and Swift [7]. Distributing compilers allow both the client and the server portion of the distributed application to be developed together. We have closely integrated RIPLEY with Volta [33, 36], a distributing compiler that splits .NET applications, translating them into JavaScript as needed.

Integration with Volta significantly simplifies the process of code replication because the entire application is given to the Volta compiler at compile time. Moreover, Volta provides a relatively narrow, standards-based API for DOM manipulation, freeing us from worrying about accurately mimicking browser-specific peculiarities within the replica. RIPLEY also integrates into the RPC infrastructure of Volta, making the process of communication between RIPLEY components across different application tiers convenient. However, the ideas of RIPLEY are fully applicable to Silverlight [37], Flash [18], as well as regular AJAX applications.

Performance and scalability implications. Another challenge of the RIPLEY approach is that it places an additional re-execution burden on the server. This is reflected in the form of CPU and memory overhead, as the replicas run on the server, as well as network overhead because of the need to transmit events.

To reduce the CPU overhead, the re-execution takes place within a client replica that runs in .NET on the server; because it is JIT-ed and the IL is strongly typed, .NET is in many cases considerably faster than JavaScript. While it is possible to run the replica within a browser on the server, this is generally prohibitively expensive if the server needs to support multiple concurrent clients: in this naïve approach, the memory overhead alone would be about 50 MB per replica. For efficiency and scalability, we run the replica within a lightweight headless *browser emulator* instead of a full-fledged browser. We are careful not to create unnecessary network traffic: we combine event transfer with existing RPC into the same network packets. While the focus of this paper is full replication, it is possible to replicate only certain, integrity-critical parts of the application to cut down the overhead.

Paradoxically, sometimes RIPLEY can even lead to *better performance*: since the application is replicated on the server, and the server is typically faster than the client, the client replica running on the server enables it to anticipate RPCs from the client in advance. This helps it to prepare and send the reply to the client ahead of time, using a push technology such as Comet[43]. In the best case, the client has the illusion of the server taking zero time for executing the RPC, leading to *zero-latency RPCs* and further improvements in responsiveness.

Contributions. This paper makes the following contributions:

- We demonstrate that replication is a practical and effective solution to the pressing problem of preserving computational integrity of distributed web applications, all without requiring developer involvement or changes to the development process.
- We propose a number of performance optimizations that alleviate the network, memory, and CPU overhead imposed by the use of replication. Surprisingly, RIPLEY may make applications more responsive: because the replica often finishes before the client, RPC results can be pro-actively *pushed* to the browser, effectively resulting in 0-latency RPCs.
- We evaluate the effectiveness and overhead of RIPLEY on five representative security-sensitive Volta applications. To give a sense of how RIPLEY might scale to larger applications, we also estimate the overhead of replicating the client-side portion of Hotmail, an existing widely-used Web 2.0 application.

Paper Organization. The rest of the paper is organized as follows. Section 2 summarizes the threat model and provides an overview of RIPLEY assurances. Section 3 gives a detailed description of RIPLEY implementation choices. Section 4 describes the results of applying RIPLEY to five security-sensitive AJAX applications. Section 5 presents a discussion of RIPLEY design. Section 6 presents related work and Section 7 concludes. Appendix A describes our benchmarks in detail, and Appendix B provides a brief formalization of integrity guarantees provided by RIPLEY.

2. THREATS AND ASSURANCES

While distributing compilers [11, 36, 46] propose a powerful programming model for distributed application development, moving execution to the untrusted client tier clearly *diminishes* the security of the resulting distributed application compared to the single-tier original [19]. It is the primary goal of RIPLEY to *restore* the level of security that has been lost. Note that RIPLEY does not try to enhance the security beyond that: a SQL injection [2] or a cross-site scripting vulnerability [5, 13] in the original application will persist in the distributed version; reliance on RIPLEY does not negate the need for input sanitization. However, with RIPLEY, we ensure that distributing the application will not *worsen* the application security posture. For instance, input sanitization checks are automatically replicated on the server, ensuring that a malicious client cannot bypass them. Replicating such checks also automatically ensures that the client- and the server-side sanitization checks are consistent with each other.

Basic assumptions. RIPLEY does not rely on the integrity of the client computation. Just as with Web 1.0 applications, the basic assumption throughout this paper is that anything executing on the server tier is believed to be un-compromised and trusted, whereas the client tier, including the browser itself, may be compromised. In practice, tampering is typically performed by either manipulating HTTP requests or editing the code executing on the client; the interested reader is referred to Hoffman and Sullivan [20] for more details about specific AJAX vulnerabilities and exploits. Furthermore, the basic replication technology is part of our trusted computing base: we assume that the translation from .NET to JavaScript provided by Volta is faithful.

Just as with a Web 1.0 application, the computation is performed based on the user-provided input (captured as an event stream). If the user is malicious and is deliberately manufacturing events, or even if the browser or the underlying OS are compromised, RIPLEY is not going to provide worse (or better) integrity compared to a fully server-side Web 1.0 application. Of course, completely breaking the need for user input in the form of events for a Web 2.0 or form inputs for Web 1.0 application is impossible if the application is to interact with the user at all.

Key goals and non-goals. The key focus of RIPLEY is to provide assurance to application developers or deployers. The key property of the RIPLEY model is that the execution that is trusted takes place *entirely* on the server. The RIPLEY server and replica pair execute based on the event stream received from the client. Ultimately, we are concerned with preserving the integrity of *persistent state* of the application, which might include database and file system operations executed by the server.

The client-side component is *only there* to enhance the responsiveness of the application. It is possible for the client-side state to deviate from the replica state; this may not be noticed until the next RPC or ever, if that difference does not affect RPCs at all. However, we are not concerned with preserving the client-state of a malicious user. If we do not wish to detect malicious users, we might choose to completely ignore RPCs we are receiving from the client in the browser, relying instead on trusted, high-integrity values computed by the replica.

RIPLEY does not eliminate the need for input sanitization, however, the ability to not worry about the placement of sanitizers illustrates the convenience of the RIPLEY model: a sanitizer check will be first performed on the client and then re-executed within the replica. So, for a benign user unintentionally supplying malformed input, the check will fail quickly on the client. It is not our goal to convince the user that a particular security policy is satisfied within the application; this is the focus on much recent work in language-based security [7, 8, 10]. Neither it is our focus to ensure that the user is communicating with the right application or that the browser or user machine are un-compromised. This can be accomplished through remote attestation methods [28]. Also, man-in-the middle attacks can be addressed with SSL.

In summary, RIPLEY ensures that a distributed web application behaves just like in the Web 1.0 case: user inputs are untrusted, but the integrity of the server-side persistent state is guaranteed.

3. RIPLEY IMPLEMENTATION

This section focuses on the implementation details of RIPLEY. In Section 3.1 we describe Volta, the distributing compiler we used

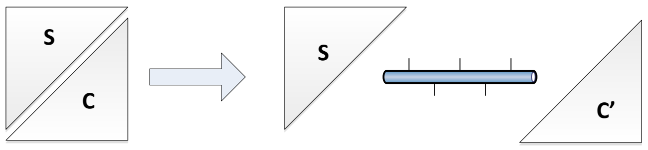

Figure 3: Tier-splitting in Volta: an application is split into a server-side component S and a client-side component C. The original .NET client-side component C is translated into JavaScript C' to be run within the browser.

in our implementation. Although RIPLEY does not have to rely on Volta to work, integrating with Volta makes our approach considerably cleaner. The techniques in RIPLEY are in fact suitable for any distributed Web application written in JavaScript [12], Silverlight [37], or Flash [18], once the client logic is mimicked on the server with the help of a replication technology such as server-side JavaScript. However, Volta allows us to both perform replication and restrict the expressiveness of the programming model to limit some of the difficult-to-replicate features; Section 5 provides a more detailed discussion of replication issues.

The remainder of the section provides a deep dive into the RIPLEY implementation. Throughout this section the reader may find it helpful to refer to the following list of components, whose relationship is shown in Figure 1:

- The server-side component S running in a .NET CLR within the web application server;
- The client-side component C' running in JavaScript within the browser;
- The replica of the client-side component C running in a .NET CLR within the web application server.

RIPLEY is implemented as an optional extension to the Volta tier-splitting process. This process takes the original application and produces S and C, optionally translating C into C' that runs in JavaScript. Integrating with the Volta tier-splitter allows RIPLEY to be implemented as several simple IL-to-IL bytecode rewriting passes. We assume that the Volta translation (RPC introduction, etc.) preserves the original application semantics. We also assume that the emulator further described in Section 3.4 is going to faithfully represent key portions of the client state such as the DOM and cookies. Given enough assumptions about the original-to-Volta program and Volta program-to-Ripley program mappings, we can for example argue that if the server is connected to an external store, such as a SQL database, running a RIPLEY-protected version of the application and a standalone version of the application will result in the same queries sent to the database. Of course, from the standpoint of the developer, enabling RIPLEY on an existing Volta application is as easy as ticking a checkbox in a Volta project configuration. In the rest of this section, we shall describe each of the components above in detail.

3.1 Volta Background

While the RIPLEY approach can be used for general AJAX-based Web applications, integrating with Volta provides a number of clear advantages. As illustrated in Figure 3, the Volta compiler is a distributing compiler that takes a .NET application as input and *tier-splits* it into a client and a server component by replacing appropriate cross-tier method calls with AJAX RPCs. Data is serialized before being sent to the server and deserialized on the server once received. A similar serialization-deserialization happens when the server returns control to the client. The client-side component is

```
// a custom button handler
this.button.Click += delegate {
    var name = this.userName.Value;
    var pass = this.passWord.Value;
    Login l = new Login();
    l.attempt(name, pass);
}
```

(a) A typical button on-click handler

```
// our rewriter adds the following handler
this.button.Click += delegate {
    // capture the event
    HtmlEventArgs evt = this.Window.Event;
    // read target object ID
    var id = evt.__ObjectId;
    // event type: keyboard, click, etc.
    var type = evt.Type;
    // extra event-specific data
    var data = serializeData(evt);

    // enqueue event for transfer
    __ClientManager.
        enqueueEvent(type, data, id);
}
```

Figure 4: (b) RIPLEY-generated handler for event interception.

translated into JavaScript for execution in an unmodified standards-compliant browser [33, 36].

Volta allows the developer to declaratively define which portion of the application runs on the server and which part on the client with the help of class-level annotations. Tier-splitting is performed subsequently as a .NET bytecode rewriting pass that reads the placement annotations, introducing RPCs as needed. To implement RIPLEY, we have augmented the Volta tier-splitter to perform additional rewriting steps described in Section 3. We have also augmented the base Volta libraries to provide support for browser emulation, as described in Section 3.4.

Volta simplifies some of the emulation challenges: because the Volta interface to the DOM is a relatively narrow, standards-based API, we do not have to worry about faithfully reproducing browser incompatibilities for every known browser in the emulator. More specifically, Volta disallows free-form HTML manipulation, exposing a well-typed DOM API that can be replicated on the server without the need for an HTML renderer or JavaScript interpreter. The alternative of checking the browser `User-Agent` string and customizing emulator behavior based on that is possible, but requires more implementation effort.

3.2 C': Instrumenting the Client

Prior to being translated to JavaScript, the client binary C generated by the tier-splitter is rewritten to capture client-side user events and send them to the server. Note that in the RIPLEY model, the server is trusted; we assume that sending events to the server does not violate the privacy of the user. If, for example, mouse click coordinates are used as a source of randomness on the client to perform client-side data encryption, sending them to the server might be undesirable, as it will undo the point of client-side encryption.

3.2.1 Event Handling

In RIPLEY, events are classified into two kinds — primitive events and custom events. *Primitive events* include each key press and mouse click event, regardless of whether the application actually has registered any handlers for them. *Custom events* are those that the application has registered explicit handlers for. A typical handler for a button click event is shown in Figure 4a. Clearly, it is crucial to intercept these events on the client and relay them to C for replay. Tracking primitive events helps maintain the state of crucial elements such as text areas and radio buttons. For instance, each keystroke a user types into an HTML form will produce a separate keyboard event that is intercepted by RIPLEY and transferred to the replica. As an optimization, multiple keyboard events sent to the same HTML element can be combined into a single meta-event.

Note that we do not handle *all* JavaScript events that occur on the client; doing so would involve listening to *all* `MouseMove` events, for example, which occur every time the user repositions the mouse pointer. Clearly, this would be prohibitively expensive and generally unnecessary. Another reason is that our DOM emulation discussed in Section 3.4 is only an approximation of the real DOM and does not maintain information about the mouse position, etc. This seems sufficient for applications we study in our experiments.

3.2.2 Event Interception

Primitive events are intercepted by registering a handler for each on the HTML `BODY` element. Since in the HTML event model, all events bubble up (or propagate) to the top-level document `BODY` element, it is a convenient point to intercept them. To intercept custom events, RIPLEY registers an extra handler shown in pseudo-code in Figure 4b for each event of interest, via bytecode rewriting. Note that we perform this rewriting at the .NET level, *before* the IL is translated into JavaScript.

RIPLEY-generated event handlers enqueue details about the event into an application-specific queue. In addition to the event type (key press, key release, etc.), the serialized event details include the key code for keyboard-related events, mouse button information for mouse events, etc. Finally, the unique identifier corresponding to the DOM object which raised this event is also sent over, so that the event can be delivered to the corresponding DOM object within the replica.

3.2.3 Event Transfer

To reduce the number of round trips to the server, which is likely to become a bottleneck on high-latency connections, events are asynchronously relayed to the server in batches. Figure 5a and 5b show two scenarios of how events may be batched on the client and transmitted to the server. There is a natural trade-off between eager and lazy event transfer. As Figure 5a demonstrates, sending events eagerly will result in excess of network usage, which might be costly on a mobile connection, for instance, but will ensure speedy replication on the server. On the other hand, batching events longer as in Figure 5b would result in minimal network usage, but will delay the integrity checking and resulting server updates and responses.

To resolve this trade-off between responsiveness and network usage, we adopt a simple middle-path strategy. Events are batched until the queue reaches the maximum size of a network packet, in which case they are sent over immediately. Otherwise, whenever there is a RPC call, all events in the queue are flushed to the server.

3.3 S: Adding a Ripley Checker

RIPLEY modifies the server binary S to receive and properly handle events arriving from the client and relay them to the client replica C for replay. Events are deserialized from the wire before being delivered to C. RIPLEY intercepts the RPCs that are received from both the JavaScript client and the replica and records them into *audit logs*, as shown in Figure 6. Note that checking is only needed if we wish to actively *detect* malicious clients — other-

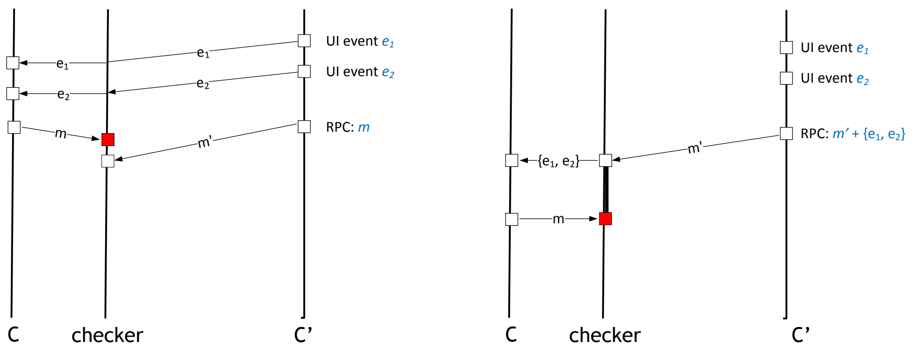

Figure 5: Eager (a) and lazy (b) event transfer. Events e_1 and e_2 arrive one after another. In a, they are sent over to the server right away. In b, they are queued-up and sent with the next RPC. While the overall completion time is later in the lazy case (as shown by a red filled square), one network message vs. three messages is used.

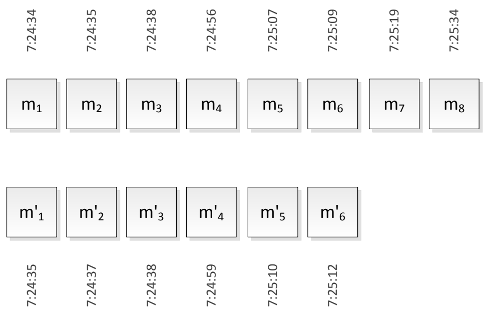

Figure 6: Audit logs from C and C'.

wise, the application can simply ignore RPCs m' received from C', replying on m values instead, with the server integrity fully preserved.

By default, RIPLEY waits until it receives and compares RPCs m and m'. Only when they are equivalent does the runtime relay the RPC call to the application server code. The return response from the server is again intercepted as a string at the HTTP level. Copies of the response are relayed to both the client replica C and the actual client C' over the network.

Lock-step execution fashion is not the only option. Alternatively, RIPLEY could allow the server-side client replica C to move ahead, by relaying m to the server and sending back the response. When m' arrives, the server can confirm its equivalence with m. This is a likely scenario with well-provisioned servers and relatively slow clients. An alternative approach consists of keeping audit logs for messages arriving from both C and C' and to do periodic randomized cross-checking offering a lower overhead at the cost of a probabilistic integrity guarantee. Moreover, if RPCs are large, sending the entire RPCs is entirely unnecessary — to save bandwidth, we can simply compute Message Authentication Codes (MAC) and send them over.

Since there could be multiple clients connected to the same server at runtime, the client replica C is executed in its own APPDOMAIN, a lightweight process-like abstraction in the .NET runtime [42]. At runtime, RIPLEY maintains a separate APPDOMAIN associated with each user session, and looks it up each time a batch of events is received from the client.

The main advantage of using separate APPDOMAINs is memory isolation: each uses its own heap and loads its own copy of dynamically linked libraries and maintains its copy of global data structures. Moreover, cross-APPDOMAIN communications are cheaper than inter-process communication in general as they do not require a process context switch and APPDOMAINs can share certain DLLs. We should point out that on a multi-core machine the RIPLEY replicas can be put on the extra cores, if those are currently unused. In this architecture, it would also be desirable to co-locate the client-side replica on the same core as the server thread it is communicating with. We further address the question of server scalability in the next section.

3.4 C: Emulator and the Client Replica

By now, one question begs to be asked: how are we going to scale a RIPLEY server? Not only are we running the existing server code, for reasons of security, we have also migrated client replicas for all clients connected to the server. Our goal of faithfully replicating the client execution on the server can be accomplished by running an instance of the actual full-fledged browser loaded with the application code on the server, one per user, as proposed by Deepfish [35]. However, for a popular and complex application, this approach is difficult to scale because the browser is a highly memory- and CPU-intensive piece of software.

There are two primary reasons that we believe that our solution will scale. First, we run the replicas in .NET instead of JavaScript, making it significantly faster in our experiments. One reason for the speed difference is that .NET has a JIT-ing highly optimized interpreter operating on a low-level strongly-typed IR. JIT interpreters for JavaScript are not as matured yet, and might still require dynamic type checks since it is not strongly typed. Second, we use a lightweight emulator instead of a full-fledged browser to reduce the memory and CPU utilization, as demonstrated in Section 4.1.

Much of the execution and state of the client does not affect the server state. For instance, any of the DOM rendering code or the state associated with the layout of the UI widgets do not feature in the application logic that updates application state on the server or the database. Clearly, such details can be abstracted away when we execute the client replica. We accomplish this by building a *browser emulator* that hosts the client replica C instead of an actual browser. The emulator is a lightweight browser that keeps track of the relevant UI state including the structure of the DOM and contents of editable elements. Since it performs no rendering or layout related computations, it avoids a lot of computation. As shown in Section 4.1, the memory footprint is an order of magnitude less for the emulator compared to a full browser.

The emulator is built as a dynamically linked library that ex-

Benchmark application	Lines of code JavaScript	C#	Frequency of remote procedure calls (RPCs) with RIPLEY	w/o RIPLEY
Shopping Cart	698,832	594	one at checkout	on every cart update
Game of Sudoku	699,873	658	one at the end	on every game cell entry
Blog Application	699,071	341	submit, load each blog	submit, load each blog
Speed Typing Test	697,782	363	initialization, finish	on every word entry
Online Quiz	699,056	416	load questions, finish	on every question

Figure 7: Summary of statistics pertaining to the RIPLEY benchmark applications.

	Application RPCs		Network overhead for event transfer					
					Uncompressed		Compressed	
Benchmark	RPCs	Bytes	Events	RPCs	Total	Norm.	Total	Norm.
Shopping Cart	1	157	13	1	1,548	119	300	23
Game of Sudoku	1	160	146	8	16,953	116	812	5.6
Blog Application	9	1,595	252	11	31,090	123	863	3.4
Speed Typing Test	4	1,598	556	28	63,945	115	1,422	2.6
Online Quiz	2	275	66	4	7,801	118	445	6.7

Figure 8: Network overhead measurements after applying RIPLEY.

poses a DOM manipulation interface, with which the client replica C links at runtime. For reasons of efficiency, in addition to using the emulator, the replica is linked against a slightly modified Volta client runtime, that relays the HTTP requests to the server component S directly using a .NET method call instead of sending it over the network.

To ensure that the replica exhibits the same *observable behavior* as an actual JavaScript client, some further machinery is required. Relaying events to the right object within the replica is done by associating each DOM node with a unique ID. Each time a new DOM node is created, either on the actual client or on the replica, a new ID is created and stored within the node. Since the runtime behavior of the actual client and its replica is identical, new DOM objects are created in the same order, providing a deterministic mapping between DOM elements of the client and its replica.

When an event is raised on a client DOM object, the ID of the target object is sent over the wire to the replica on the server, as shown in Figure 1. The APPDOMAIN hosting the replica maintains a lookup table of IDs-to-object references, which allows RIPLEY to identify the appropriate object instance to deliver the event to. The method to be invoked on that instance and the parameters that need to be sent are provided as part of the event.

4. EXPERIMENTAL RESULTS

A comprehensive evaluation of RIPLEY presents a serious challenge: interesting third-party security-sensitive benchmarks applications are hard to come by. While our reliance on Volta makes implementation easier, it makes finding good benchmarks harder, since none of the existing Volta applications are security-critical. To get a better sense of runtime overhead RIPLEY induces, we used a combination of five representative security-relevant applications we have developed using Volta, as described in Section 4.1, which we evaluate in detail with a focus on *absolute* overheads. In Section 4.2, we show the overhead of applying RIPLEY-style replication to Hotmail, a large-scale AJAX application. We show RIPLEY overhead *relative* to the resource and CPU requirements of existing Hotmail code.

Volta benchmarks. Our Volta benchmark applications and the benefits of applying RIPLEY to them are described in detail in Appendix A. A brief summary of information about them is given in Figure 7. All of these applications were originally developed in C# and (partially) translated into JavaScript by the Volta compiler. Columns 2 and 3 provide the line-of-code metric for the resulting JavaScript and original C# code. Note that the JavaScript code includes the translated versions of the required system classes that may be needed at runtime, which causes it to be quite substantial; if GWT experience is any indication, we expect code size to decrease drastically in subsequent Volta releases [30]. Column 4 shows the frequency of RPCs in the version of the application protected with RIPLEY. In most cases, there is only one RPC required at the end of the execution: in a way, we were trying to push the limit of client-side computation without excessive server interaction. Many of our benchmarks were inspired by existing game exploits found in the book "Exploiting Online Games" [21].

Comparing with hand-secured applications. To put the benefits of automatic replication into perspective, for each of our benchmarks we also consider an application that would have the *same strong integrity properties* written by hand or with the help of a compiler such as Jif [7, 8]. In the majority of cases, engineering such an application requires manually moving significant portions of the computation to the server to preserve integrity. Column 5 shows the number of RPCs for such an application. Clearly, RIPLEY results in fewer RPCs for the same integrity guarantee. This is particularly important when extra network round-trips come at a high premium, such as in the case of mobile phone environments.

4.1 Volta Applications: Overhead Micro-measurements

We focus on three dimensions of overhead: extra network utilization, extra memory utilization, and extra CPU time. This section focuses on micro-measurements of performance overhead.

4.1.1 Network Overhead

The network overhead was measured for each application by performing a pre-determined series of user events for each, mirroring typical use of that application. The amount of network usage in terms of the stream of events sent to the server was measured in each case, using Firebug [29]. The first group of columns, columns 2–3, in Figure 8 shows the network usage of the application itself. Most applications in our benchmark suite send only a few RPC messages to the server. The Blog application has been

	Volta	RIPLEY
Server	23 – 26	27 – 32
Client (IE)	59 – 64	59 – 65
Client (FF)	69 – 77	69 – 78

Figure 9: Comparison of memory utilization, measured in MB.

written to produce one RPC per blog entry read, and so it uses more messages than other applications.

Columns 4–7 show the network overhead introduced by using RIPLEY. The "Total" column shows the total number of bytes and the "Norm." column shows number of bytes per event. Extra network activity is only due to transmission of event data to the server. Unsurprisingly, applications such as the Speed Typing and the Blog that generate a lot of key strokes consume more network resources. However, network messages containing event data are sent asynchronously and thus do not significantly slow down the client-side execution. The bandwidth requirement is directly proportional to the number of events, as can be seen in the last column. All applications use up about 120 bytes per event, uncompressed.

Fortunately, the event stream is highly compressible; applying GZip compression reduces the size of a single event on the wire to just 3–4 bytes on average, as shown in columns 8–9. The effectiveness of compression is most noticeable in highly interactive benchmarks such as Speed typing, reducing the number of needed network packets to just a single one in most cases. Unfortunately, the current generation of browsers do not support automatic compression of HTTP requests, only HTTP responses, requiring it to make compression part of Volta tier splitting, which is part of future work.

4.1.2 Memory Overhead in the Emulator

In the presence of multiple replicas running alongside the server it is possible for the replicas to use up quite a bit of extra memory. Of course, the emulator is significantly less memory-consuming than running a full-fledged version of the browser.

To experimentally demonstrate this point we first considered a version of the Shopping cart application running without RIPLEY and then with RIPLEY enabled, with both Internet Explorer and Firefox running on the client. A summary of information about this experiment is shown in Figure 9. The table shows the range of memory utilization, in megabytes, to support a single client for each version with and without RIPLEY; in most cases, more memory was allocated as the application progressed. We used Internet Explorer version 7.0.6001 and Firefox version 2.0.0.16 on Windows Vista to perform these measurements. The server memory utilization goes up by about 5 MB by adding the RIPLEY emulator. The experiment was conducted thrice to verify that the increase is always about 5 MB. This is *an order of magnitude* cheaper than adding a full-fledged browser with a memory footprint of over 50 MB.

Furthermore, we modified the server to create more client replicas to simulate the process of a multitude of clients that are simultaneously connected to the server. Figure 10 shows the server memory size as we increase the number of replicas to 100. Because of DLL sharing across the different APPDOMAINs, the marginal cost of an additional replica is only about 1.3 MB compared to 5 MB.

To put this into perspective, we have consulted recent capacity planning guidelines for Microsoft Sharepoint, a sophisticated Web 2.0 application providing wiki and document sharing functionality. In many ways, Sharepoint is a good application to consider: unlike search, which is server-bound, it performs a reasonable amount of computation on the client, but the reason to have client-side computation is responsiveness, not to unload computation from the server. Capacity planning guidelines suggest 100–190 concurrent users per machine [1,47]. At 1.3 MB per user, this requires 130–247 MB of extra memory per server, which is a reasonable overhead.

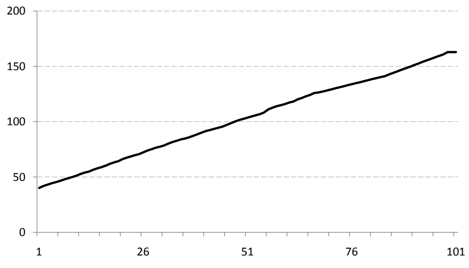

Figure 10: Memory footprint, in MB, as a function of the number of replicas.

	Server checks			Event capture			
Benchmark	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Med**
Shopping Cart	0.083	0.083	0.083	8	0	1.21	1
Game of Sudoku	0.462	0.462	0.462	87	0	1.25	1
Blog Application	0.079	0.002	0.012	8	0	0.676	1
Speed Typing Test	0.078	0.004	0.023	84	0	0.8	1
Online Quiz	0.078	0.004	0.041	162	0	3.044	1

Figure 11: CPU overhead in ms after applying RIPLEY.

4.1.3 CPU Overhead

RIPLEY introduces CPU overhead on both the server and the client. Clearly, running the replica on the server also consumes CPU resources, despite the fact that running within the emulator described in Section 3.4 makes things considerably faster.

RIPLEY checking on the server introduces some latency for regular requests, as shown in columns 2–4 in Figure 11. The server runs an ASP.NET application server on a dual-core 3-GHz machine with 4 GB of RAM running Microsoft Vista. We run each application with the same input as for the network overhead experiments and measure the time that each client-side request spends waiting for the replica to generate the corresponding request and to compare the two to verify its integrity. In most cases, the former component forms the bulk of the overhead, since the replica receives the event information in batches and needs time to catch up with the actual client.

The maximum overhead of the Shopping Cart and Sudoku is due to this effect, since the events are sent to the replica right before the `checkout` and `finishgame` RPCs are sent to the server. The maximum overhead for the other applications is observed during the application initialization phase and typically involves application-specific IO on the server. For instance, in the Blog application, it involves fetching the blog data and in the Quiz application, it involves initializing the database of questions. Despite this, this overhead is mostly negligible for all benchmarks. The minimum times were fractions of milliseconds, since for these requests, the replica is already in sync when the requests arrive. The overhead is only due to the string comparison of messages.

Client-side instrumentation for capturing and serializing event information to the server adds execution overhead in the browser. The overhead is low as shown in columns 5–8 in Figure 11, typically about a couple of milliseconds on average. The extremes of minimum and maximum are shown to indicate the spread. The high extremes are believed to be statistical anomalies since the median overhead is 1 ms for all applications, which is not noticeable for interactive GUI applications. This is the typical overhead we might expect, since the events are sent asynchronously to the server. Moving event capture to the browser as discussed in Section 5 is likely to reduce the client-side CPU overhead even further.

4.2 Hotmail Experiments: Overhead Macro-measurements

The experimental evaluation thus far has focused on studying several relatively small Volta applications in much detail. To provide an estimate of how RIPLEY ideas might extend to an existing large scale Web 2.0 AJAX application, we have devised a series of experiments focusing on replicating the client-side Hotmail state "by hand", without automatic deployment provided by RIPLEY for Volta applications.

4.2.1 Hotmail Experimental Setup

The Hotmail mailbox in our setup consisted of 32 email messages, about half of which were HTML-heavy. The entire application download with a clean browser cache consists of 793 KB of both code and data, uncompressed. Much of that — 703 KB total — is JavaScript. When pretty-printed, the downloaded JavaScript adds up to 31,178 lines.

Pretty-printing is necessary, because often, to save space, JavaScript is shipped as a long line of code. Note that the DOM representing the mail application UI is partially downloaded from the server and partially created on the client, through JavaScript client-side logic and `XmlHttpRequest` calls.

4.2.2 Network Overhead

To measure the network overhead of event capture, we loaded a Hotmail inbox window and waited for network activity to quiesce. For this run, the mailbox contained a total of 8 unread email messages and several dozen read ones. We used Hotmail for a total of 5 minutes, starting by reading the messages, replying to mail, deleting spam, cleaning the junk mail folder, searching the mailbox for and removing some large HTML emails, etc.

This entire experiment produced a total of 491 keyboard and mouse events (subsequent keyboard presses sent to the same HTML element were compressed into one event), captured using a key logger. When saved to disk, the event trace was 8,673 bytes or 2,889, when compressed with GZip (for this experiment we used exact screen coordinates instead of element ids, leading to a slightly more verbose representation). Next, we examined the amount of traffic that exists between Hotmail servers and the browser.

For this experiment, we ignore traffic generated by viewing email messages to other hosts: for instance, viewing a message from Amazon.com results in numerous requests for book cover images, etc. We also discount requests that the browser makes to extension vendor sites, etc. At the end, we end up with a total of 328 requests, some going directly to Hotmail servers, others going to affiliated advertising sites.

The aggregate size of the requests is 617,297 bytes; the aggregate size of responses is 3,045,249 bytes. Part of the reason HTTP requests are generally larger than expected is because they contain sizable tracking cookies: cookies of 1–2 KB in size are not uncommon. The extra network traffic because of event capture is negligible in comparison: it constitutes 1.4% if we do *not* compress the event stream and 0.4% otherwise.

Page	**JavaScript heap**
Hotmail homepage	0.815 MB
Inbox page	1.25 MB
HTML-heavy email	1.291 MB

Figure 12: JavaScript live heap sizes for Hotmail.

4.2.3 Memory Footprint

Client-side state, which is what our replica needs to keep track of efficiently, consists of three major components. First, there is the DOM displayed in the browser. The DOM is not *displayed* in the RIPLEY emulator, but its representation needs to be kept in memory. Second, there are non-DOM client-side data structures maintained in the JavaScript heap. Third, there is the JavaScript code maintained on the client. Our assumption is that within the replica, this code will be represented in .NET and shared across the instantiated replicas, so that the code footprint does not grow as more concurrent clients are added. Therefore, we focus on measuring the amount of per-client DOM and heap state.

DOM state. To estimate the DOM state within the replica's heap, we have taken a saved version of the DOM as HTML. We converted this 20 KB HTML document into a DOM representation within the emulator by recursively parsing the HTML and recreating the DOM tree on the .NET emulator heap. The in-memory representation of the DOM is 349 KB, if we enable frequent garbage collection in .NET to simulate memory pressure, and 434 KB otherwise.

Client-side heap. Unfortunately, translating the existing 700 KB or so of JavaScript into .NET is not a tractable approach, so instead we have chosen to instrument the runtime execution of Hotmail within a real browser to put an upper bound on the amount of heap storage likely to be required within the replica.

For this experiment, we instrumented the JavaScript engine in Mozilla Firefox to report the aggregate size of live heap objects after each garbage collection cycle. (Again, we forced frequent garbage collection to simulate memory pressure in a multi-user environment. As with any garbage collection environment, while a lot of objects are *allocated*, a high fraction of them die young; we focus on the aggregate size of live heap objects on the heap.) Note that this is an upper bound because .NET, being a strongly-typed runtime environment, generally uses the heap more efficiently than JavaScript.

We have discovered that even an empty browser allocates a fair bit of state — 3.1 MB in total — in the JavaScript heap, which includes browser plugin state, JavaScript built-in functions, etc., all reflected into JavaScript. None of this needs to be represented in the replica, so this heap size constitutes the baseline for our experiments. In other words, we focus on heap size increase seen when we run Hotmail within Firefox.

Memory footprint numbers measured in megabytes for different Hotmail pages in our experiment are shown in Figure 12. For none of the frequent actions, does the JavaScript heap size exceed 1.3 MB. When considered together with the DOM state, this is still under 1.75 MB per connected client.

4.2.4 CPU Overhead

The CPU overhead for replicating the client-side of Hotmail is quite difficult to measure accurately, given that the browser is busy with so many activities at once. Just as with memory, we would have to translate the existing Hotmail code into .NET and run it. However, it has often been observed that JavaScript execution is *not* the dominant part of client-side application execution, in terms of the fraction of time it takes. For instance, a study of 100 top sites suggests that JavaScript execution is responsible for about 3% of overall time. For a JavaScript-heavy site such as Hotmail, the reported JavaScript time is only about 14%, when run within Internet Explorer. On the other hand, CSS, layout, rendering, DOM manipulation, and data marshalling take a total of 75% [48]. Luckily, the emulator is "headless": it does not need to worry about these issues. As such, we do not expect that replicating the client-side of Hotmail will dramatically add to the CPU load on the server.

To support our belief, we used a proxy-based JavaScript instrumenter to parse and add time-stamps in the beginning and end of event handlers found in JavaScript for Hotmail. The majority of event handlers in our experiment take a very small span of time, below the 15 ms granularity of the JavaScript built-in timer. Drilling into the execution of a single action — clicking on a large, HTML email message and waiting for it to show up in the email viewing pane — reveals the following. Event handlers themselves are very fast; heavy-weight network-bound operations execute asynchronously. For instance, an inbox message open operation is processed asynchronously. The event handler itself takes only 15 ms. Email message processing callback takes 125 ms, much of which is HTML screen repainting. Again, the headless emulator does not suffer this rendering performance hit.

While it is difficult to *precisely* estimate the effect of replicating Hotmail onto the server, replication is not entirely unprecedented in the AJAX world. Replication is often used in mobile browsers to achieve compatibility. Early success of projects that use server-side rendering include the Deepfish mobile browser project [35] and a recent SkyFire browser [45], both of which rely at least partially on server-side rendering, is encouraging for the feasibility of the RIPLEY approach on a large scale.

4.2.5 *Summary*

In summary, while at over 31,000 lines of code Hotmail is a considerably larger and more complex AJAX application than the five Volta applications studied in the previous section, the overhead imposed by replication seems acceptable. The network overhead appears to be just 0.4%. The memory overhead is around 1.75 MB per concurrently connected client, which is manageable given typical capacity planning requirements of supporting 100–200 concurrent users. While is it difficult to precisely ascertain the CPU overhead of replication, is is clear that a headless browser results in a considerable overhead reduction.

5. DISCUSSION

This section considers implementation and deployment issues that arise in the design and implementation of RIPLEY. Section 5.1 discusses difficulties of faithful replication. Section 5.2 addresses scalability issues.

5.1 Difficulties of Faithful Replication

As was pointed out earlier, the fact that Volta provides a narrow browser-independent interface to the DOM makes replicating it faithfully relatively easy. Disallowing HTML injection through direct `innerHTML` assignments allows us to fully mediate access to the DOM by the application, at the cost of the code being slightly more verbose.

The most pronounced issue that makes replication hard is the presence of various forms of non-determinism. Isolating them all is a tricky problem. Below we discuss ways to introduce deterministic replay [6, 39] into replica execution with the help of additional instrumentation. The following sources of non-determinism are most common in web applications.

Using the `Random` family of functions. JavaScript exposes a random number generator through function `Math.Random`. Clearly, unless additional measures are taken, the value returned by calls to this function on the client and the replica will disagree. A uniform approach to treating randomness is to perform the computation on one, "canonical" tier. In this case, we can instrument the client-side code to block on `Math.Random` in the same way it would block on an RPC. Then the client would request the random number from the server and we would return the same value to both the client and the replica. Note that this value needs to originate on the server tier to be unguessable by a malicious client. In environments where random numbers need to be generated frequently on the client, the quality of the random number could be traded off for reduced network usage by only asking for the randomizing seed occasionaly.

Reading and measuring time. Access to time is provided through the `Date` object in JavaScript. Similarly to the approach described above, access to time routines can be instrumented and the replica can be blocked until the time measured on the client is delivered to continue the computation.

Accessing third-party servers. A systematic way to deal with client-side code accessing third-party servers is to require that these accesses be tunneled through the server. For servers in a different domain, this is necessary anyway, because of the same origin policy in JavaScript. This allows for easy centralized access to outside data, for both the replica and the client-side code. Because calls to external services are performed only once, this also deals with the issue of non-idempotent calls with side-effects. We also assume that the client-side code does not interact with other JavaScript, because replicating that JavaScript is far from easy. In other words, *all* client-side JavaScript code is generated by Volta.

Scheduling issues. Unlike many other languages, JavaScript provides a single-threaded execution environment. However, there are still opportunities for non-determinism caused by the use of timers (functions `setTimeout` and `setInterval`), which allow scheduling a piece of code to run on a timer. Other projects have focused on deterministic replay of multi-threaded software [6]. While logging and replay of event processing order is possible, for simplicity, we currently disallow timers in RIPLEY.

Offline storage. Our emulator supports a cookie-based data store. The emulator faithfully replicates the data stored in the browser. While we have not experimented with other forms of offline store such as Gears [14], we believe it can be supported similarly. RIPLEY provides a nice model for offline execution: both events and RPCs are buffered up before the client is reconnected, at which point the entire offline client-side execution is replayed within the replica. Offline persistent store can also be supported by RIPLEY if the user does not have privacy objections to sending their data to the server.

Browser Enhancements. In fact, a set of small changes to the JavaScript interpreter would help us secure event capture and delivery and would also address the sources of non-determinism discussed above. In particular, instrumenting `Math.Random` and `Date` routines as well as event handlers as described in Section 3.2 *in the interpreter* is the easiest and most systematic way to treat these issues that ensures that malicious JavaScript code co-existing within the same page is unable to gain access to this data. Event capture outside of JavaScript should result in a lower overhead.

Plugins an other client-side code. RIPLEY is not designed to interoperate with plugins that might be running within the page. In fact, the RIPLEY model discussed in this paper is targeting standalone deployment of a RIPLEY application witnin an HTML frame;

allowing other code to co-exist within the same frame as part of a mash-up, for example, can easily compromise agreement with the replica. This is because co-located script can change both global data structures as well as code on the client-side.

5.2 Performance and Scalability

The RIPLEY model enables the following interesting optimization opportunities.

0-latency RPCs. An advantage of the RIPLEY architecture is that, once computed, RPC results can be actively pushed to the client [43]. This way, when the RPC is finally issued on the client, its result will already be available, leading to 0-latency RPCs. This form of pre-fetching demonstrates that not only does RIPLEY make the application more secure, in many cases it can also make it more responsive, especially for CPU-intensive workloads.

MAC-ing RPCs. To further reduce the network overhead we may send MACs (message authentication codes) of RPCs m' instead of their actual values.

Deployment strategy. RIPLEY meshes nicely with the traditional load-balancing approach to deployment of large-scale Web 2.0 applications. In particular, a load balancer could be used to repeatedly direct the same user to the server where both its replica and the corresponding server threads run. Currently, this functionality is implemented in the RIPLEY checker, which looks up the appropriate APPDOMAIN for a user session. Moreover, to save memory, both the server thread *and* the replica can be serialized on high server load for long-running sessions and then brought back from disk.

Dependency analysis. An important observation is that the entire client-side code base does not have to be included in the replica. In particular, display code does not need to be executed on the server because the replica is essentially "headless" — there is no user to see the GUI. To further reduce the amount of code the replica must run, we can use a slicing analysis [51] to only include a portion of the client-side code that contribute to values included into RPCs. This is left as future work.

6. RELATED WORK

The security of the Web infrastructure has been a subject of much previous work. The various approaches to solving the problem can be categorized roughly along four lines of inquiry. A sizable body of literature has focused on the static analysis of web applications using techniques such as taint-checking. Runtime monitoring of web applications has also proved to be effective. Others have addressed the problem at a higher level by developing a cleaner and more secure programming model, often erasing the boundaries between various tiers. Recent work has also developed techniques to protect against untrusted clients in a networked environment. Finally, the idea of security through replication has also been well studied in earlier work. We elaborate further on each of these.

6.1 Analysis and Monitoring

There has been a great deal of interest in static and runtime protection techniques to improve the security posture of traditional "Web 1.0" applications [23, 26, 31, 34, 52]. Static analysis allows the developer to avoid issues such as cross-site scripting prior to deployment. Runtime analysis, on the other hand, allows exploit prevention and recovery.

The WebSSARI project pioneered this line of research. WebSSARI uses combined unsound static and dynamic analysis in the context of analyzing PHP programs [23]. Several projects subsequent to WebSSARI improve on the quality of static analysis for PHP [26, 52]. The Griffin project proposes a scalable and precise sound static and runtime analysis techniques for finding security vulnerabilities in large Java applications [31, 34]. Static analysis is also used to drastically reduce the runtime overhead in most cases. The runtime system allows vulnerability recovery by applying user-provided sanitizers on execution paths that lack them. Several other runtime systems for taint tracking have been proposed, including Haldar et al. for Java [17] and Pietraszek et al. [41] and Nguyen-Tuong et al. for PHP [40]. All these techniques can be used in conjunction with RIPLEY, by applying them on the complete code base that includes the client and the server subprograms.

While server-side enforcement mechanisms are applicable for traditional web applications that are composed entirely on the server side [26, 31, 52], Web 2.0 applications that make use of AJAX often fetch both data and JavaScript code from many sources, with the entire final HTML only available within the browser, making runtime client-side enforcement a natural choice.

Recently, there has been a number of proposals for runtime enforcement mechanisms to ensure that security properties of interest hold for rich-client applications executing within the browser [10, 22, 25, 56]. Erlingsson et al. make an end-to-end argument for the client-side enforcement of security policies that apply to client behavior [10]. Their proposed mechanisms use server-specified, programmatic security policies that allow for flexible client-side enforcement, even to the point of runtime data tainting. Unlike RIPLEY, their technique can enforce some necessary, but not sufficient conditions for establishing distributed application integrity.

Guha et al. [16] propose an analysis to construct a model of valid client-side behavior and a security monitor that rejects client requests that fall outside that model. This approach provides a form of partial integrity, defeating important classes of XSS and CSRF attacks. While it is able to impose a validity requirement on client-side requests, it is unclear how well this approach applies to arbitrary data structures exchanged between the client and the server, when data integrity is the problem. Also, this approach may suffer from false positives. In contrast, RIPLEY's approach is admittedly much more blunt: it does not try to form an approximation of valid behavior, it just runs the program and compares the end-result, providing a sufficient condition for integrity.

6.2 Web Programming Models

Tier-splitting has been proposed in settings other than Volta as a way to program distributed web applications. Popular systems in this space include Links [11], Hop [46], Hilda [54], etc. To the best of our knowledge, RIPLEY is the first realistic security solution for these kinds of frameworks.

BASS is a recent attempt to build security into a declarative high-level web programming model, working on the observation that security issues are often orthogonal to the main web application logic [55]. It enables the programmer to specify the business logic of the application without needing to write the security related logic. Abstractions for common operations, such as form input, are baked into the model. Secure coding practices that prevent common attacks such as CSRF, XSS and session fixation are applied by the language compiler. A prototype implementation of the translation exists, but no applications seem to have been written in BASS. RIPLEY, on the other hand, is a realistic programming model integrated with a full-fledged Volta compiler. Instead of protecting only against common exploits, RIPLEY defends against any client attack that attempts to compromise application integrity. BASS does not deal with client-side scripting at all, whereas RIPLEY works in a model where a significant portion of the application is run on the client for enhanced responsiveness.

6.3 Untrusted Clients

Protection against untrusted clients and eavesdropping over the network has received much attention, especially in the context of online gaming [21, 53]. In a distributed online game, part of the application workload is typically delegated to the clients and the server keeps track of only an abstract state of the game environment. As a result, the game is rendered vulnerable to malicious clients compromising the physical and logical rules governing the simulation in the game. Hacking popular online games is a financially viable undertaking as game "items" can be converted to real-world currency or sold on eBay.

Jha et al. propose a solution to the distributed online game integrity problem by performing random audits of the client state verifying that the client has not manipulated its state in violation of the semantic rules of the game [24]. Our approach, in contrast, provides a non-probabilistic guarantee of integrity at a potentially higher cost. In particular, if the client-side computation is highly CPU-intensive, as ray-tracing in games tends to be, despite replying on an emulator and running in a faster .NET environment, with sufficiently many connected clients, the RIPLEY server might eventually become overwhelmed.

6.4 Replication & Replay for Security

Replication is a well-known way to increase security assurance, previously studied in file systems and replicated state machines [4, 32, 44, 49]. The work closest to ours is that of Zheng et al. [57–59]. In many ways a precursor to Swift [7, 8], this work focuses on splitting programs while conforming to a set of integrity and privacy policies. The latter are addressed by computing in the hash space, not unlike our Quiz application described in Appendix A.

A high-level difference in philosophy with our work is that we avoid using annotations, believing that having to write annotations places an undue burden on the developer. For instance, one such approach requires about 20–30% of program lines to be annotated [8]. This makes such techniques challenging to retrofit into existing unannotated code. Instead, we "blindly" replicate the entire client-side portion of the program on the trusted server tier, using runtime optimizations to make this approach scalable. In doing so, we trade increased developer productivity for runtime overhead.

Beyond our main focus on computational integrity violations caused by malicious users, RIPLEY may also address the situation of a benign user placed in a "malicious environment". This could be a propagating JavaScript worm, such as Samy [50]. Additional RPCs issued by the worm on behalf of the user will not be generated by the replica as it runs in .NET. This discrepancy will be spotted by RIPLEY, disallowing worm propagation. Replication will also prevent cross-site scripting attacks that result in server RPCs, because the extra injected JavaScript code is ignored by the replica executing in .NET.

Revirt [9] is a system logger that records events in the operating system so that they can be replayed later for post-mortem analysis of attacks. The OS runs on a virtual machine and the logger runs below the level of the virtual machine, thereby making it resistant to kernel attacks. The logger is also able to replay the complete instruction-by-instruction execution of the virtual machine so that any queries about its execution can be answered to detect anomalous behavior. Capo [38] is an advancement on the techniques of Revirt, enabling various styles of logging of systems on multi-processor machines. RIPLEY also allows a *complete* replay of the execution of untrusted code and is guaranteed to detect anomalous behavior. The RIPLEY logger now runs as JavaScript, but we could imagine moving it to the browser to enhance the integrity of logging, as done by Revirt.

7. CONCLUSIONS

This paper presents RIPLEY, the first fully automated approach to ensuring integrity of distributed web applications. To demonstrate the efficacy of RIPLEY in practice, we have applied RIPLEY to five realistic AJAX applications. The performance overhead introduced by RIPLEY was minimal, in terms of CPU, memory, and network overhead. While we have demonstrated our ideas in the context of the Volta compiler, the ideas of code replication can be easily extended to other runtime environments such as Silverlight or server-side JavaScript.

We believe that in the future the approach pioneered by RIPLEY may become an important building block of trustworthy distributed applications. Our work closely follows the secure-by-construction philosophy of building application software. In particular, we envision RIPLEY becoming an integral part of the next generation of application servers. All the application developer will have to do to obtain the integrity-preservation benefits of RIPLEY, is to "drop" their web application into the application server, with automatic replication becoming part of the deployment process.

Acknowledgements

We thank Jeffrey Van Gogh and Danny Van Velzen for helping us extend the Volta system, and Erik Meijer for his continued support for research efforts based on Volta. We also thank Úlfar Erlingsson, Andrew C. Myers, David Evans and Cedric Fournet for their detailed feedback on this paper. The comments by the CCS reviewers were also very helpful in shaping the final version of the paper.

8. REFERENCES

[1] J. Allspaw. *The Art of Capacity Planning: Scaling Web Resources*. O'Reilly Media, 2008.

[2] C. Anley. Advanced SQL injection in SQL server applications, 2002.

[3] J. Boutelle. Bandwidth savings with AJAX. `http://www.jonathanboutelle.com/mt/archives/2006/01/bandwidth_savin.html`, Jan. 2006.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[5] CGI Security. The cross-site scripting FAQ. `http://www.cgisecurity.net/articles/xss-faq.shtml`.

[6] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.

[7] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the Symposium on Operating Systems Principles*, pages 31–44, Oct. 2007.

[8] S. Chong, K. Vikram, and A. C. Myers. Sif: enforcing confidentiality and integrity in Web applications. In *Proceedings of USENIX Security Symposium*, pages 1–16, Aug. 2007.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 211–224, Dec. 2002.

[10] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web application security. In *Proceedings of the USENIX Workshop on Hot topics in operating systems*, pages 1–6, May 2007.

[11] Ezra Cooper and Sam Lindley and Philip Wadler and Jeremy Yallop. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, Oct. 2007.

[12] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 4th edition, 2002.

[13] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[14] Google Gears. `http://gears.google.com`.

[15] Google Web toolkit. `http://code.google.com/webtoolkit`.

[16] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proceedings of the International Conference on World Wide Web*, pages 561–570, Apr. 2009.

[17] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference*, pages 303–311, Dec. 2005.

[18] A. Hartman. *Exploring Adobe Flash CS4*. Delmar Learning, 2009.

[19] B. Hoffman. Ajax security dangers. `http://www.spidynamics.com/assets/documents/AJAXdangers.pdf`, 2006.

[20] B. Hoffman and B. Sullivan. *AJAX security*. Addison-Wesley Professional, 2007.

[21] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley Professional, 2007.

[22] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the USENIX Workshop on Hot topics in operating systems*, pages 1–7, May 2007.

[23] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the International Conference on World Wide Web*, pages 40–52, May 2004.

[24] S. Jha, S. Katzenbeisser, and H. Veith. Enforcing semantic integrity on untrusted clients in networked virtual environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 179–186, May 2007.

[25] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International Conference on World Wide Web*, pages 601–610, May 2007.

[26] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, May 2006.

[27] A. Judson. Tamper data Firefox add-on. `https://addons.mozilla.org/en-US/firefox/addon/966`.

[28] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the USENIX Security Symposium*, pages 21–21, Aug. 2003.

[29] R. Lerner. At the forge: Firebug. *Linux J.*, 2007(157):8, 2007.

[30] B. Livshits and E. Kiciman. Doloto: Code splitting for network-bound Web 2.0 applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 350–360, Sep 2008.

[31] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the USENIX Security Symposium*, pages 18–18, Aug. 2005.

[32] D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *Proceedings of the The IEEE Symposium on Reliable Distributed Systems*, page 51, Oct. 1998.

[33] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Software*, 25(5):53–59, Oct. 2008.

[34] M. Martin, B. Livshits, and M. S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, 2006.

[35] Microsoft Corporation. Microsoft Live Labs Deepfish. `http://labs.live.com/deepfish/`, 2006.

[36] Microsoft Corporation. Microsoft Live Labs Volta. `http://research.microsoft.com/~emeijer/CloudProgrammability.html`, 2007.

[37] Microsoft Corporation. Silverlight. `http://silverlight.net`, 2007.

[38] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceeding of International Conference on Architectural Support for Rrogramming Languages and Operating Systems*, pages 73–84, Mar. 2009.

[39] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Computer Architecture News*, 33(2):284–295, May 2005.

[40] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, pages 372–382, June 2005.

[41] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection*, Sept. 2005.

[42] J. Richter. *CLR via C#*. Microsoft Press, 2006.

[43] A. Russell. Comet: Low latency data for the browser. `http://alex.dojotoolkit.org/wp-content/LowLatencyData.pdf`, 2006.

[44] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Compututing Surveys*, 22(4):299–319, Dec. 1990.

[45] S. Segan. For Skyfire's mobile Web, the secret's in the server. `http://www.pcmag.com/print_article2/0,1217,a%253D223932,00.asp`, Jan. 2008.

[46] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *Companion to the Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 975–985, Oct. 2006.

[47] S. Smith. Capacity and performance planning. In *European Microsoft SharePoint Conference 2007*, Feb. 2007.

[48] C. Stockwell. What's Coming in IE8. `http://blogs.msdn.com/ie/archive/2008/08/26/ie8-performance.aspx`, 2008.

[49] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised system. In *Proceedings of the Conference on Symposium on Operating System Design and Implementation*, pages 12–12, Oct. 2000.

[50] The Samy worm. `http://namb.la/popular`.

[51] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[52] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, Aug. 2006.

[53] J. Yan. Security design in online games. In *Proceedings of the Annual Computer Security Applications Conference*, page 286, Dec. 2003.

[54] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven Web applications. In *Proceedings of the International Conference on Data Engineering*, pages 32–43, Apr. 2006.

[55] D. Yu, A. Chander, H. Inamura, and I. Serikov. Better abstractions for secure server-side scripting. In *Proceeding of the International Conference on World Wide Web*, pages 507–516, Apr. 2008.

[56] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 237–249, Jan. 2007.

[57] S. Zdancewic and A. C. Myers. Secure information flow and CPS. *Lecture Notes in Computer Science*, 2028:46–61, 2001.

[58] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of Symposium on Operating System Principles*, pages 1–14, 2001.

[59] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 236–250, May 2003.

APPENDIX

The appendix section is organized as follows: Appendix A provides a description of our benchmarks. Appendix B outlines a formalization approach for RIPLEY guarantees.

A. BENCHMARK APPLICATIONS

In this section we describe the benchmark applications we used to test RIPLEY. All of these applications have been developed on top of Volta. A summary of information about these applications is given in Figure 7. In our choice of applications, we were inspired by previous application benchmarks [7] as well as attacks presented in books "Exploiting Online Games" [21] and "AJAX Security" [19]. We particularly wanted to demonstrate how the AJAX model allows us to migrate virtually the entire computation to the client, with a minimum of RPCs, and how, when RIPLEY is applied, this approach does not lead to a reduction in integrity.

A.1 Client-side Shopping Cart

As in a typical shopping cart within an e-commerce application, one can add and remove items to the cart, update their amounts, and eventually check-out. There is a provision for using coupons values where designed coupons `C5`, `C10`, and `C15` denote 5%, 10%, and 15% discounts, applied to the cart total.

As described in Section 1, it is typical for such an application to carry out the total calculation on the server side, which means that every cart update results in a RPC to the server. Our shopping cart is implemented entirely on the client, with only one message containing the cart total sent to the server upon check-out.

Security threats: In many ways, the shopping cart application is a demonstration of typical client-side security threats described in Section 2. For example, a malicious user may attempt to manipulate the discount computation by using invalid coupons or using the same coupon multiple times. Alternatively, the malicious client can just manually set the resulting total before it is sent over to the server without even touching the code.

Benefits of RIPLEY: With RIPLEY enabled, we can afford to do these computations on the client side, thereby preserving the application responsiveness. Since the user events are replicated on the server side, the server also maintains an abstract state of the cart, which includes values of various form fields, and can easily verify the total amount as soon as it is received from the client.

A.2 Game of Sudoku

This online game presents one of five hard-coded Sudoku puzzles for the user to solve. The solution is checked on the client and sent over to the server to be recorded for computing user ratings, etc. As the game progresses, there are two kinds of validation checks being performed. After a number is typed into a game cell, the row and the column is checked to look for repetitions; repeated numbers are flagged in red. When the user is ready to submit a solution, the entire grid is checked for validity.

Security threats: Both the local and the global validation checks of the game state can be easily bypassed by a malicious user, leading them to declare the puzzle as finished without making an effort.

Benefits of RIPLEY**:** When the result of the game is submitted to the server, RIPLEY will check the validity of the final solution based on the event stream that it receives as input. A single RPC may be used to submit all the relevant client-side events at once without creating extra network traffic.

A.3 AJAX Blog

This online blog application allows the user to view a blog, and to post and edit blog entries.

Security threats: Unlike the previous two applications that address the issue of a malicious client, the focus here is on protecting the *benign* client from the effects of script injection and worm attacks. By default, the blog application does not perform extensive data sanitization, leaving itself open to cross-site scripting attacks. Worms can be used to amplify the effects of cross-site scripting. In the case of a blog, a worm may post a blog entry on behalf of an unsuspecting user.

Benefits of RIPLEY**:** In case of a JavaScript worm, when the worm tries to propagate by uploading executable contents to the server, it will do so by sending extra RPCs. Because the client replica runs on the server side in .NET, it is impervious to JavaScript code injection. As a result, the mismatch in the stream of RPCs will be detected by RIPLEY. Also, client-side checks can now be reliably performed on the client.

A.4 Speed Typing Test

In this application, a set of words is randomly chosen from a dictionary and displayed to the user as a paragraph. The objective for the user is to type as many words as she can within the time limit of one minute. The user's word-per-minute count and accuracy is calculated once the time limit has passed. As the words are typed in, their correct spelling is checked and highlighted on the fly. An interesting property of this application is that events arrive at a very rapid rate, thereby stressing the performance side of RIPLEY.

Security threats: A malicious user may tamper with per-word spelling checks and also manipulate the time measurements to further rig the test.

Benefits of RIPLEY**:** Just like in previous applications, RIPLEY ignores circumvented checks on the client, performing its own computation within the replica.

A.5 Online Quiz

In this quiz application, trivia questions appear one by one, and depending on the correctness of the current answer, the next question is selected, of a higher or same point value, respectively. After answering a total of ten questions, the user's score is calculated and sent to the server for recording. The answer to each question consists of a single word.

In an online quiz application such as this, the answer would be sent to the server for checking after each question and the next question would be returned. This requires a round trip after every question, making the application less responsive. If the quiz is timed, the round trip overhead needs to be properly taken into account. In contrast, our design moves the entire database of questions (62 questions total) to the client. The next question's selection is performed on the client, only requiring a single RPC at the end.

Security threats: An interesting twist in the Quiz application compared to the ones above is that the confidentiality of the data on the client is important. Indeed, if the client can easily learn and enter the proper answers, cheating on the quiz would be trivial. In general, RIPLEY does not do anything to address confidentiality concerns, relegating these concerns to the developer.

For this application, we use a simple confidentiality-preserving approach. We only send hash values of the proper answers instead of the answers themselves. This allows us to compare hash values of the provided answers with the correct ones. We chose to allow for one-word answers to each question instead of multiple-choice. This way we make dictionary attacks, while not impossible, then considerably more difficult; they would be trivial if the space of answers were small, consisting of choices A–E. Additionally, just as for the applications above, a malicious client can manipulate the solution checking code and related data.

Benefits of RIPLEY**:** As the entire application is run on the client side, integrity issues like bypassing solution checking, etc. can be handled by Ripley as the checks are replicated on the server side. RIPLEY cannot address confidentiality concerns in general, though.

B. RIPLEY GUARANTEES

In this section we briefly outline the integrity preservation property that we intuitively described in Section 2. A full formalization is subject for future work and will likely require modeling Volta applications in a small, well-controlled language that is easy to reason about. For a given non-distributed program P_O, there are three separate server-based versions of P_O to consider:

- P_O: Original Web 1.0 program that has not been tier-split;
- P_S: Tier-split program C+S with C running within a full-fledged browser *on the server*; and
- P_R: RIPLEY-replicated program C+S with replica C running within a RIPLEY emulator.

Let us emphasize that all these three variants of P_O are executed *entirely on the server*, only relying on input from the user. Let us further assume that the program is connected to permanent database store, and this store is the only form of persistent state the application maintains. We are interested in ensuring that this store cannot be affected by malicious client actions. Let us further assume that the programs above do not have the forms of non-determinism described in Section 5. We can then show the following.

Property: For any run of P_O and P_R given a stream of input events $\langle e_1, e_2, \ldots, e_n \rangle$, the sequence of SQL database calls issued by P_O and P_R are exactly the same.

This property protects the integrity of persistent server-side state: starting with the same database state, we will end up with the same database state for both P_O and P_R. This is what we mean when we say that RIPLEY restores the integrity guarantees to their Web 1.0 state. Given the same set of (untrusted) inputs communicated to the program as well as assumptions of determinism, both the RIPLEY and the original application will arrive at the same answer.

Proof of the Property above proceeds by showing that there is a bi-simulation between P_O and P_S and another bi-simulation between P_S and P_R. This transitively makes P_O and P_R bi-similar, which in particular implies the Property.